Key-Value Stores

BDNR · Non-Relational Databases
M.EIC · Master in Informatics Engineering and Computation

Sérgio Nunes
Dept. Informatics Engineering
FEUP · U.Porto

Based on Chapter 8 from NoSQL Distilled, Sadalage and Fowler (2012)

Outline

- Key-Values Stores
  - Concepts
  - Features
  - Use Cases
- Redis
- Support in Postgres

Key-Value Stores

Key-Value Store

- Key-value store are part of the simplest NoSQL solutions from an API perspective.
- The client can [set , get] the value for a key, or [delete] a key.
- The value is opaque to the data store (i.e. it is not exposed).
  - Note that some key-value stores offer complex data structures (e.g. Redis).
- The application is responsible to understand and manipulate the data stored.
- Given the simplicity of the model (key-based access pattern), the result is:
  - very high performance and
  - easy scalability.

Key-Value Concepts

- Each data record corresponds to a **key-value pair**.
- **Keys** are unique and provide access to each pair.
  - Keys can be application defined or generated by the DBMS.
  - Examples include: session ids, hashes, URIs, filenames, timestamps.
- **Values** represent data with arbitrary data types, structure, and size.
  - Serialization/Deserialization of data is the responsibility of the client application.
- In practice, implementations exhibit additional features over this core set.

Key-Value Solutions

- Oracle Berkeley DB, www.oracle.com/database/berkeley-db (1994) [GNU] (embedded)
- Memcached, memcached.org (2003) [BSD]
- Redis, redis.io (2009) [BSD]
- Riak, riak.com (2009) [Apache License]

- Amazon DynamoDB, aws.amazon.com/dynamodb (2012) [Proprietary]
- Microsoft Zure Cosmos DB, azure.microsoft.com/en-us/services/cosmos-db (2017) [Proprietary]

- More: db-engines.com/en/ranking/key-value+store

Key-Value Stores Characteristics

Consistency

- Consistency is applicable only for operations on a single key.
- In distributed key-value stores, eventually consistency is adopted.
- Optimistic writes — i.e. let conflicts occur then correct them — are difficult to implement because record values are opaque to the data store (thus cannot know if they have changed).
- Conflict resolution strategies based on timestamps are more common, e.g. adopt the latest value.

Querying

- Key-value stores offer key-based querying (i.e. sole access path).
- The data is opaque to the data store, thus querying by data attribute is not possible.
  - Note that implementations offer alternative querying features, e.g. with additional indexes.
- The design of the keys needs to be carefully thought.
  - In relational systems its good practice to adopt 'meaningless' keys decoupled from the data domain.
  - In key-values stores, meaningful keys, that can be obtained from user input or generated, are important to provide data access (e.g., user ID, session ID, date-based).
- Example key naming convention:
  - [Entity Name : Entity Identifier : Entity Attribute] ==> "Customer:12345:FirstName"

Writing

- Atomicity is guaranteed at the individual key-value pair.
- Implementation of multi-operation transactions varies between data stores.
- Some key-value stores have full transaction support.

- A common feature in key-value stores are transient keys, i.e. keys that expire.
  - E.g., after a time interval, or at a given time.
  - Commonly used for caching setups.

- Key-values stores commonly operate in-memory, but most offer persistency mechanisms (e.g. periodic dumps).

Data Model

- Key-value stores have a schemaless model.
- Values can be text, blobs, JSON, etc.
- Redis differs since it supports several built-in data types.

Scaling

- Sharding is a common option for scaling in key-value stores.
- With sharing, the value of the key determines the node to which the key is stored.
- Primary-replica replication is also common in key-value stores for reliability.

Use Cases

Suitable Use Cases

- Caching
  - In-memory solution to improve performance in read-only configuration.
  - Caching layer between the main database and the application code.
- Storing Session Information
  - User sessions are usually identified by a unique value (session id).
  - Storing each user session information under this key make (all) session information access very efficient (store or retrieve), using a single operation.
- User Data
  - Access to an aggregate with all user preferences can be done by a unique username or id.
  - Examples include: user settings, shopping carts, etc.

When Not to Use

- Characteristics that pose problems to key-value solutions.
    - Relationships between data items (e.g. navigate between records, relational data);
    - Correlate data between different items (e.g. obtain aggregate results or use range queries);
    - Transactions over multiple key operations;
    - Access items by data attribute;
    - Operations over multiple keys at the same time;
    - Large individual records (that don't fit in memory).

- As always, although these are deterring characteristics, a deeper analysis is needed to properly balance tradeoffs in the design of the system, as these problems can be solved on the client side or with other complementary technologies.

Key-Values Store Summary

- Records are access by single unique keys.
- Records are not related to each other.
- Very fast for random key-based access (fast).
- Scalable with easy data partitioning.
- Simple (schemaless) data model and programming interface.
- Client applications are responsible for modeling and manipulation of data values.
- Client applications need to know the key for data access (i.e. the only access path).

Redis

Redis

- redis.io
- Redis is an open source, in-memory data structure store.
- Redis provides complex data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, etc.
- Redis has built-in replication, transactions, and different levels of on-disk persistence, and offers automatic partitioning.
- Features also include: transactions, publish-subscribe, scripting, transient keys.
- A wide range of support for other programming languages is available.

Redis Software

- **redis-server** is Redis server itself
- **redis-cli** is a command line interface (CLI) utility to interact with a Redis server.

- Redis download are available at: redis.io/download
- Official Docker images exist at: hub.docker.com/_/redis

Main Data Types

- String, are the simplest data type and can contain any type of data.
- List, series of ordered values.
  - Top or bottom elements are very fast to retrieve.
  - The order is preserved.
- Hash, a map between a string field and a string value.
- Sets, series of unique, non-ordered values.
  - Direct access to any element is fast (e.g. test for membership).
  - Only one element per set can exist.
- Sorted Sets, similar to a regular set but each element has an associated score.

Basic Operations

- Write key-values with the **SET** command:
  - SET user:name "John Smith"
- Read values with the **GET** command:
  - GET user:name => "John Smith"
- Test if a key exists with the **EXISTS** command:
  - EXISTS user:name => 1
  - EXISTS user:address => 0
- Keys can be deletes with the **DEL** command:
  - DEL user:name

Keys

- With the command **KEYS** you get a list of all defined keys.
  - KEYS *
  - KEYS users:*
- The command **TYPE** provides information about the key data type.
- The command **FLUSHALL** deletes all keys.

Counters

- Keeping counters are a common use case for Redis.
- The commands **INCR and INCRBY** can be used to increment the value of a key.
  - SET requests 0
  - INCR requests => 1
  - INCRBY requests 10 => 11
- The commands **DECR and DECRBY** are used to decrement a value.
  - DECR requests => 10
  - DECRBY requests 5 => 5

Transient Keys

- Keys can be defined to exist for a certain period.
- The **EXPIRE** command defines the length of time of a given key (in seconds).
  - SET cache:score "1-1"
  - EXPIRE cache:score 300 (5 minutes)
- The expiration time can be set in the SET command.
  - SET cache:score "1-1" EX 300
- The time to live of a given key can be checked with the **TTL** command.
  - TTL cache:score => 294
- An expiration configuration can be disabled with the **PERSIST** command.
  - PERSIST cache:score

List Operations

- List elements are ordered.
- Elements can be added to lists with the **RPUSH and LPUSH** commands.
  - RPUSH people "Alice"
  - RPUSH people "Miguel"
  - LPUSH people "Maria" "Rui" (multiple values can be added)
- Lists elements can be listed with the **LRANGE** command, with the start and offset as parameters (-1 to the end of the list, -2 to the penultimate, etc).
  - LRANGE people 0 -1 => "Rui", "Maria", "Alice", "Miguel"
  - LRANGE people 1 2 => "Maria", "Alice"
- Other list commands include:
  - **LLEN** (list length), **LPOP** (remove and return from the start), **RPOP** (remove and return from the end).

Set Operations

- Sets are unordered.
- Elements can be added to sets with the command **SADD**.
  - SADD colors red
  - SADD colors purple yellow
- Set members can be removed with **SREM** or listed with **SMEMBERS**.
  - SREM colors purple
  - SMEMBERS colors => "red", "yellow"
- Other set commands include:
  - **SISMEMBER** (test for existence), **SUNION** (combine multiple sets and return values).

Sorted Set Operations

- With sorted sets, each element has an associated score.
- This score is used to sort elements.
- The **ZADD** command adds elements to a sorted set with the respective score.
  - ZADD monuments 330 "Eiffel Tower"
  - ZADD monuments 93 "Statue of Liberty"
- Sorted elements can be retrieved with the **ZRANGE** command.
  - ZRANGE monuments 0 -1 => 1) "Status of Liberty", 2) "Eiffel Tower"

Hash Operations

- Hashes are maps between string fields and string values.
- Adequate to represent objects.
- The **HSET command** is used to set hash values.
  - HSET user:101 name "Pedro"
  - HSET user:101 country "Portugal"
- The **command HGETALL** is used to retrieve all values for a key.
  - HGETALL user:101 => "name", "Pedro", "country", "Portugal"
- Or get a single field with the **command HGET**.
  - HGET user:101 country => "Portugal"

Postgres Support

Key-Value Store Support in Postgres

- Postgres supports key-value storage with the hstore data type.
- Keys and values are text strings.
- Keys are unique and the order of the pairs ir not significant.Persistent solution with
- Model abstraction is the main advantage.
- Scalability is still dependent on Postgres scalability features.
- Documentation: www.postgresql.org/docs/current/hstore.html

Key-Value Store Support in Postgres

```
CREATE EXTENSION HSTORE;

CREATE TABLE mytable (h hstore);

INSERT INTO mytable VALUES ('a=>b, c=>d');

SELECT h['a'] FROM mytable;
 h
---
 b
(1 row)


UPDATE mytable SET h['c'] = 'new';
SELECT h FROM mytable;
          h
----------------------
 "a"=>"b", "c"=>"new"
(1 row)
```

Questions or comments?

References

NoSQL Distilled
Pramod J. Sadalage and Martin Fowler
Addison-Wesley, 2012

Next Generation Databases
Guy Harrison
Apress, 2016

Seven Databases in Seven Weeks
Lic Perkins
The Pragmatic Programmers, 2018